Application for United States Patent

of

Steven Edward Atkin

for

Bidirectional Domain Names

CROSS-REFERENCE TO RELATED APPLICATIONS

(CLAIMING BENEFIT UNDER 35 U.S.C. 120)

Not applicable.

FEDERALLY SPONSORED RESEARCH

AND DEVELOPMENT STATEMENT

This invention was not developed in conjunction with any Federally sponsored contract.

MICROFICHE APPENDIX

Not applicable.

INCORPORATION BY REFERENCE

Not applicable.

BACKGROUND OF THE INVENTION

Field of the Invention

[0001] This invention relates to the arts of computer and information displays for multiple languages, alphabets, and scripts. The invention relates especially to the arts of human interfaces (input, display, output) concerning computer network and World Wide Web addresses in languages which require bidirectional display and presentation.

Description of the Related Art

[0002] The World Wide Web and the Internet have become everyday technologies in most developed economies, and is now becoming an integral part of the process in developing economies. Its ability to communicate information, both in written form such as web pages, graphic form such as photos and videos, and data form such as extensible markup language (XML), is becoming a key factor to every industry in every country of the world.

[0003] However, the current technologies supporting the World Wide Web are "English-centric" due to the roots of the beginning of the Internet being an American and European effort. As such, many of the conventions and "standards" employed in servers, routers, e-mail protocols, etc., employ an English alphabet with English-like syntax. Initially, companies and individuals in non-English countries were able to adopt and use these technologies due to their ability to work in both their native language and English.

[0004] It is, though, possible that certain information and concepts cannot be mapped into English from a native language, and thus represents an inability of the English-centric World Wide Web (WWW) to effectively communicate this information and these concepts. Further, the successfulness with which consumers may "find" a business on the WWW depends on their ability to input or select a web address which is logical and rational. If a business has a native language name, there may not be a logical or rational English equivalent. As such, businesses which primarily deal in non-English marketplaces may find their success in "going online" less than optimal given that they must adopt an English domain name.

[0005] Unicode's ability to represent multilingual text makes it good candidate for establishing the basis for a domain name structure. Unicode brings not only an encoding framework, but also support handling display requirements such as bidirectional scripts. The collection of Unicode's character equivalences is both desirable, and at times necessary, given Unicode's goal of encoding natural language text. These equivalences, however may present problems in the context of domain names.

[0006] Unicode's BiDirectional (Bidi) algorithm may be unsuitable for determining an appropriate display ordering for domain names. Specifically, the Bidi algorithm itself possesses a set of implicit assumptions about the usage of common characters. This set of assumptions may not be applicable to domain names. Domain names use the same repertoire of characters that appear in text. This requires a different algorithm for handling domain names.

[0007] The transition from the now ubiquitous monolingual ASCII based domain name system to a truly multilingual extendable system has been long awaited. Indeed, it may have already begun without waiting for standards to be developed. This move brings the goal of realizing a multilingual World Wide Web one step closer. Nevertheless, this transition must be approached cautiously as decisions made today may have long lasting effects. These decisions include the set of characters for constructing names, the base character encoding, and the codepoint transmission protocol.

[0008] There are, however, certain constraints that must be observed, regardless of these decisions. For example, domain names that are "legal" today must still remain legal in the new domain name system, otherwise the new system will not receive widespread acceptance. It is impractical to expect a vast overhaul or retrofit of thousands or millions of content servers, domain name servers, and routers in order to support a new, non-backwards-compatible domain name system.

[0009] A likely starting point for choosing the allowable set of characters from which domain names may be constructed is to start with the character repertoire available in the well-known Unicode/ISO10646 standard. The range of characters available in Unicode is vast and accommodates most modern written scripts. In contrast to ASCII, it includes scripts such as Arabic, Farsi and Hebrew.

[0010] At first glance, extending the current domain name system may not seem to be much of a challenge, given that all that needs to be done is to add more characters to the script. However, unlike ASCII, which only encodes scripts written and displayed in a left-to-right order, Unicode encodes scripts written right-to-left, as well as those written left-to-right. Additionally, in Unicode, it is perfectly "legal" to intermix these scripts, which provides not only for a wider variety of single-language displays, but also for displays of mixed content. However, when these scripts are intermixed, their display may become ambiguous, due to the conflicting directions.

[0011] In creating a new domain name system, such ambiguities must not exist. The display of such domain names can not simply be left up to the user or application software, which would certainly lead to confusion.

[0012] In order to alleviate this situation, a BiDirectional domain name method and system must not allow for ambiguities in the interpretation, display, or analysis of a BiDirectional domain name. Additionally, this method and system must be both simple to understand, easy to implement, and inexpensive to execute, in order to facilitate its widespread acceptance and use.

[0013] Therefore, there is a need in the art for a system and method which allows domain names to be handled and displayed with different (non-English) reading orders. Further, there is a need in the art for this system and method to be readily usable within the currently deployed technologies of the World Wide Web, and compatible with existing methods and systems such as Unicode's BiDi algorithm.

BRIEF DESCRIPTION OF THE DRAWINGS

[0014] The following detailed description when taken in conjunction with the figures presented herein provide a complete disclosure of the invention.

[0015] Figure 1 shows an example of two different logical encodings of a Latin capital letter "u" with diaeresis and acute, which should ideally represent in the same display order.

[0016] Figure 2 shows an example of characters which are glyph variants of other characters, namely the "fi" sequence.

[0017] Figure 3 illustrates issues involved with interpreting and displaying typical domain names using the well-known Unicode BiDi algorithm, and the ambiguity which can result.

[0018] Figure 4 illustrates similar ambiguities which may arise when interpreting strings containing the hyphen-minus "-" character, also of significant importance to the display of multilingual domain names.

[0019] Figure 5 shows an example of ambiguities which arise when a label within a domain name contains characters with varying directions.

[0020] Figure 6 illustrates a string reversal operation where digits are treated as units.

[0021] Figure 7 sets forth the well-known organization of computer platforms such as web browser computers, web servers, and personal computers.

SUMMARY OF THE INVENTION

[0022] The system and method of the invention provide for unambiguously determining the display order of multilingual domain names. The system and method implements the correct semantics of the full stop "." or "dot" character, as well as the hypen-minus "-" character as used within domain names, and provides a one-to-on logical order to display order conversion function. Each domain name is broken into a plurality of individual labels separated by the full stop character, and each individual label is independently evaluated for proper bidirectional display order. The resulting mapping of logical order to display order provides unambiguous resolution of multilingual domain names.

DETAILED DESCRIPTION OF THE INVENTION

[0023] In accordance with the objectives of the present invention, a primary goal of a multilingual domain name display and analysis system and method is to unambiguously represent multilingual domain names. Optimally, the multilingual domain name system and method must also:

(a) provide a one-to-one mapping between names in logical order and names in display order;

(b) be consistent with Unicode's Bidirectional Algorithm when possible;

(c) be easy to understand and simple to implement;

(d) absent of requirement for any form of normalization; and

(e) minimize impact to the current DNS architecture.

[0024] The present invention is preferably realized as a function or method in computer-executable software for Internet servers, clients, and routing devices. Turning to FIGURE 7, a generalized organization of such a computer platform (70) is shown. The computer platform (70) has a central processing unit (CPU) (74), a set of device drivers and a basic input/output system (BIOS) (78), and typically an operating system (703), such as Linux, UNIX, or Microsoft Windows [TM].

[0025] Most computer platforms, such as a personal computer, are also equipped with disk interfaces (75) and disks; user device I/O (76) to interface to keyboards, pointing devices, and a display; and a network interface card or device (77) allowing communications to a computer network, wireless network, or the Internet. Some computer platforms, such as personal digital assistants, web-enabled telephones, and Internet appliances may not be provided with all of these components, but in general, the functionality of these components is present in some form.

[0026] The computer platform (70) is also typically provided with one or more non-portable, machine-specific application programs (702).

[0027] According to the preferred embodiment, the computer platform (70) is provided with a Java interpreter (701), which are freely available for a variety of operating systems and computer platform, and which are well-known in the art.

[0028] The remainder of this disclosure is given with respect the logical methods to be realized in computer executable software, and preferably realized in a Java program, applet or servlet.

[0029] When domain names are interspersed within natural language text the problem of displaying the text and domain names becomes rather complex. This complexity, however can be managed if the problem is broken into separate and distinct phases. The problem with simply modifying the Unicode Bidirectional Algorithm to accommodate domain names is it makes an already complex algorithm even more difficult to manage.

[0030] The essence of the Unicode Bidirectional Algorithm is first to perform contextual analysis on the text and then determine where the boundaries of the directional runs are. The general problem with this strategy is that as technology continues to expand greater and greater demands will be placed upon the bidirectional algorithm to always correctly render any and all textual data causing the algorithm to be in a constant state of flux.

[0031] When Unicode performs contextual analysis on text it overrides the static proprieties assigned to some of the characters. Specifically this occurs during the processing of weak and neutral types. Separating this portion of the algorithm from resolving implicit levels and reordering levels greatly extends the applicability of the algorithm. Ideally the analysis of the text should be distinct from the actual determination of directional boundaries.

[0032] During the analysis phase, domain names, mathematical expressions, phone numbers, and other higher order data elements are detected. Nevertheless, it is impossible to create an algorithm that can always correctly identify such elements. The real issue is whether or not it is possible to create an algorithm that identifies such elements within some reasonable range of error and under a set of acceptable constraints for the elements themselves.

[0033] The determination as to whether a stream contains a domain name is rather straightforward if the domain name is preceded by some special identifier. Specifically, “http://”, “ftp://”, or “telnet://”. When these identifiers are not present, however the ability to recognize a domain name becomes greatly diminished. The authors believe it is unreasonable to force every domain name to be preceded by some special signal. There are many cases where it is inappropriate to specify the protocol. For example, consider the case where a marketing memo specifies the name of some website.

[0034] For better understanding of the present invention, following conventions are used in examples throughout this disclosure, as patent disclosures themselves do not lend themselves well to BiDirectional and non-English script text:

(a) uppercase letters "A" through "M" are used to represent Arabic letters,

(b) uppercase letters "N" through "Z" are used to represent Hebrew letters,

(c) digits "0" (zero) through "4" are used to indicate European numerals, and

(d) digits "5" through "9" are used to indicate Arabic numerals.

[0035] This representation is summarized in Table 1, which is the same convention used by Unicode to discuss the input and output of the Unicode Bidirectional Algorithm.

Table 1: Bidirectional character mappings

Type	Mapping
L	a-z
AL	A-M
R	N-Z
AN	5-9
EN	0-4

[0036] The richness of characters available in Unicode is certainly an asset when used to encode natural language text. Nevertheless, this richness is something that is not necessarily desirable when encoding domain names. The various ways in which characters can be constructed in Unicode, both "precomposed" and "decomposed", makes the representation of domain names unnecessarily complex.

[0037] This complexity presents two significant problems for encoding domain names: domain name registration and domain name equivalence. Historically, these have not been a problem, because it made no difference whether the registration of a domain name was based upon characters or codepoints. In ASCII, there is no distinction between characters and codepoints, however in Unicode such a distinction becomes necessary at times.

[0038] In Unicode, characters that contain diacritic marks may be represented in two ways: "precomposed" form and "decomposed" form. Characters in precomposed form are represented by a single codepoint, while characters in decomposed form are constructed from multiple codepoints.

[0039] For example, the Latin capital letter "u" with diaeresis and acute can be encoded in three different ways (10, 11, 12) in Unicode, as shown in Figure 1. In all cases, the same visual output or display is produced irrespective of the sequence of codepoints.

[0040] This has a significant impact on the clear representation of data and especially for domain names. If domain names are registered by characters and not by codepoints, then domain name servers and/or client systems will be required to perform some form of normalization.

[0041] Alternatively, if domain names are registered via codepoints and not by characters, then normalization becomes a non-problem. On the other hand, it forces the registration of multiple names (equivalent encodings) that really represent the same name.

[0042] Further complicating matters is the fact that Unicode also encodes some characters that are merely glyph variants of other characters. This situation also requires some form of normalization. For example, the two character sequence "fi" may be represented in two ways (20, 21) in Unicode, as shown in Figure 2. The first encoding (20) uses a single codepoint, while the second encoding (21) uses two codepoints. In either case, both sequences encode the same semantic content, with the only difference being the glyph used to render the sequence.

[0043] In a first aspect of the present invention, a method is adopted in which only decomposed characters are used in cases where there are no corresponding precomposed characters. This significantly simplifies and facilitates the task of determining name equivalence, as each domain name has a unique representation under this scheme.

[0044] In a second aspect of the present invention, characters which are glyph variants of other characters (compatibility characters) are not allowed to be used in domain names. While at first this may seem too restrictive, it is nothing more than an artificial restriction. In reality, there is no need for compatibility characters, as domain name distinction or equivalence is not based upon visual appearance. Further, these characters are unnecessary for legacy data conversion as they are encoded in ASCII now.

[0045] In a third aspect of the invention, control codes are excluded from the new multilingual domain names as they currently are today. These include the bidirectional controls, as well, such as LRE, LRO, LRM, RLE, RLO, RLM, and PDF. The purpose of these controls is to override the behavior of Unicode's Bidirectional Algorithm. In most situations, Unicode's Bidirectional Display Algorithm produces acceptable results when rendering natural language text. The use of these controls is only required in the rarest of situations, and thus their elimination outweighs any potential benefits.

[0046] Naturally, the set of allowable domain name characters must expand to include Arabic and Hebrew letters, however Unicode has many codepoints for the Arabic writing system and the Hebrew writing system. Not all of these code points are required in the context of domain names.

[0047] In another aspect of the present invention, a number of Arabic characters are safely excluded from multilingual domain names, including the Arabic presentation forms, UFB50-UFDFF and UFE70-UFEFC. It is safe to exclude these characters, as they only represent ligatures and glyph variants of the base nominal Arabic characters. Additionally, the Arabic points U064B-U0652, U0653-U0655, and U0670 may also be excluded. In most cases, the Arabic points are only used as pronunciation guides. If the points were to be included, then names that differed only in their use of points would be treated as if they were distinct and different names. This is not unlike the English homonyms "read" and "read", which are ambiguous. Removing the Arabic points eliminates such problems, with the understanding that not every Arabic word would be able to be uniquely represented.

[0048] Further, the Koranic annotation signs U06D6-U06ED can also be eliminated from domain names, as they are not used to distinguish one name from another.

[0049] In Hebrew, the cantillation marks U0591-U05AF and Hebrew points UFB0-U5C4 are preferrably excluded from multilingual domain names, as they are predominately used as pronunciation guides and for indicating the underlying structure of text. Additionally, the Arabic and Hebrew punctuation characters are also excluded from domain names, as they are currently not permitted or used in domain names. Table 2 summarizes the list of acceptable Arabic and Hebrew characters.

Table 2: Acceptable Arabic and Hebrew characters

Unicode Range	Script	Notes
U05D0-U05F4	Hebrew	ISO8859-8
U0621-U064A	Arabic	ISO8859-6
U0660-U0669	Arabic	Arabic-Indic digits
U0671-U06D3,U06D5	Arabic	Extended Arabic letters

U06F0-U06FE	Arabic	Persian, Urdu, and Sindhi

- -

[0050] Unicode's ability to intermix the various script systems of the world makes the creation of multilingual documents no more difficult than the creation of monolingual documents. This new found freedom, however does come with a cost. When various script systems are intermixed, their display may become unclear.

[0051] Unicode provides an algorithm for determining the appropriate display order given an arbitrary sequence of characters in logical order. The algorithm is based upon a set of implicit heuristics along with a set of explicit control code overrides. These control codes are used in cases where the implicit rules do not yield an appropriate display order.

[0052] One may naturally assume that since Unicode characters are going to be used in domain names, then Unicode's Bidirectional Algorithm should also be used. Upon closer examination, it becomes apparent that this approach or assumption is inappropriate. The input to Unicode's algorithm carries with it a set of assumptions. The primary assumption being that the input is natural language text in general. This assumption, however is not necessarily true in the case of domain names. A domain name does not resemble a paragraph of multilingual text. So, different assumptions regarding the content of a multilingual domain name must apply. This contextual difference causes several problems when one attempts to apply the Unicode Bidirectional Algorithm to domain names.

[0053] The first problem to be encountered is the use of the full stop character, U002E, within domain names. When a full stop occurs in natural language text, its purpose cannot be immediately determined. The meaning of the full stop is dependent upon the context in which it is used. It may indicate the end of a sentence, an abbreviation, or even a floating point number, such as specified in rules W4 and W5 in Unicode Standard Annex #9.

[0054] When a full stop, however, is present in a domain name, its meaning is clear. The meaning of the full stop never varies across domain names. The full stop always serves to separate a domain name into its individual parts or “labels”. Furthermore, the full stop establishes the hierarchy of the individual labels. In domain names, there is a strict hierarchy regarding the ordering of the labels. The most general part of the domain name is always the rightmost label, while the most specific part of the name appears as the leftmost label (e.g. "subdomain" labels). This requires a domain name to be read and interpreted in a general left-to-right, English-like direction.

[0055] When the Unicode Bidirectional Algorithm’s rules are applied to text, it is done on a per paragraph basis. Each paragraph is rendered independently of each other. Unfortunately, when the Unicode Bidirectional Algorithm is applied to domain names, each domain name label is not rendered independently of the others, and as such, each domain name label may influence the rendering of the others. To solve this problem, the method of the present invention implements the full stop character as if as if it were the start of a new paragraph in the context of domain names. Additionally, each domain name is rendered in an overall left-to-right reading direction so as to preserve domain name label hierarchy.

[0056] The Unicode Bidirectional Algorithm determines the general reading direction of a paragraph in one of two ways. The first method is based upon a higher order protocol explicitly stating the reading direction; the second method makes use of an implicit rule whereby the first strong directional character determines the overall reading direction. In this context the term “strong” indicates a character that is either a left-to-right character or a right-to-left character. This implicit rule, however causes problems for rendering domain names, as shown in Figure 3.

[0057] The first display (30) of Figure 3 is a "normal" domain name in presented logical order. The corresponding output from the Unicode Bidirectional Algorithm (32) shows how the presence of an Arabic character in the first label forces the entire domain name to incorrectly be given a right-to-left reading or display. This unfortunately mangles the hierarchical structure of the domain name, making it impossible to universally determine which label is the most specific and which is the most general. This, of course, would result in user difficulties reading and understanding a Universal Resource Locator (URL) or domain name, and certainly would be impossible to correctly route hyper text transfer protocol (HTTP) messages, file transfer protocol (FTP) transactions, and the like.

[0058] Some may argue that if the overall reading direction is known, in this instance right-to-left, then the hierarchy of the individual labels can be determined. This statement is not true in multilingual domain names, however.

[0059] In many cases it is impossible to tell the overall reading direction by merely looking at the output. It turns out that it is impossible to obtain the same output "display order" given two distinct inputs in logical order. In this example, the "normal" input (30) and an alternative input (31) produce the same output or display (32). In this case, the most specific part of the name of the "normal" input (30) "ABC", while in the alternative input (31) is "ibm". This does not indicate that there is a flaw in Unicode's algorithm, rather it only further illustrates the impact of the assumptions concerning the intended use of the Unicode Bidirectional Algorithm.

[0060] Normally, in natural language text processing, this is not a problem given that the two orderings can be distinguished by their physical justification on the screen, either right or left. This factor, however, is not available to domain name displays. When a domain name appears in printed text, there is no generally accepted way to indicate the overall reading direction.

[0061] Nonetheless, some may argue that if the entire domain name is in Arabic, then the label hierarchy should be reversed. The problem in adopting this strategy occurs when the entire domain name is not from the same script, as is the case in this example. The method of the invention provides a more desirable multilingual output (4) as illustrated in Figure 3, wherein the "ABC" label is a right-to-left language component of the domain name, and the "ibm" and "com" labels are left-to-right components of the multilingual domain name. This output is consistent with the current structure of domain names. In this case the full stop characters are ignored, and the bidirectional algorithm is applied to each of the individual labels of the domain name.

[0062] One might assume that Unicode's Bidirectional Algorithm may still be appropriate if it is run independently on each of the individual labels. This strategy also presents problems, however. The problem with this approach involves the use of the hyphen-minus character "-", U002D. In the Unicode Bidirectional Algorithm, the hyphen-minus is assigned to the European Terminator character class. Unfortunately, this causes the character to behave as if it were an European numeral when adjacent to European numerals, as specified in rule W5 in Unicode Standard Annex #9.

[0063] This behavior may be acceptable when processing natural language, but is unacceptable when processing multilingual domain names. In multilingual domain names, the predominant usage of the hyphen-minus is as white space, and not as an European terminator, as illustrated in Figure 4. A single domain name label in logical order (40) is presented, with the same label shown in display order (41) which is the output of the Unicode Bidirectional Algorithm. If the hypen-minus characters are treated as white space characters consistent with their use in domain name, the third display order (42) is obtained. Evident from this example is the fact that the Unicode Bidi algorithm is inappropriate for yet another reason for displaying multilingual domain names.

[0064] Yet another problem occurs when an individual label contains characters with varying directions. In this situation, the reading order of a label may become ambiguous, as illustrated in Figure 5, wherein an individual label is shown in display order (50). Unfortunately, there are two possible readings "logical order" associated with this output (51 and 52). According to another aspect of the present invention, it is assumed that in such a mixed case, a label always takes a general left-to-right reading, which allows only one possible reading. This policy is consistent with the overall left-to-right reading of a domain name. Nevertheless, the Unicode algorithm still maps the two logical inputs to the single display output even when the overall reading direction is fixed to left-to-right. This situation potentially causes problems for domain name resolution.

[0065] Further according to the present invention, multilingual domain name registration is made in logical order. This policy is consistent with how bidirectional data is generally stored in files today. If the Unicode Bidirectional Algorithm is permitted to be used for the display of domain names, then there may be situations when a domain name can not be resolved even when it appears to be entered correctly. One possible solution to this situation is to register multiple logical names that yield the same display order. However, according to the method of the invention, a better approach is provided which eliminates this many-to-one relationship. Using the method of the invention, each display order is mapped to one and only one logical input, and each logical input is mapped to one and only one display output. This policy comes with some associated cost, however. There maybe cases where the reading may seem unnatural. It is believed, however, that this will occur infrequently and that the benefits outweigh any potential misreading.

[0066] The method of the invention is divided into two phases: inferencing and reordering. Inferencing resolves the direction of indeterminate characters, such as the full stop or "dot", hyphen-minus, Arabic numeral, and European numeral. During this phase each character is assigned a strong direction, either left or right. The reordering phase takes the fully resolved characters and generates a display ordering for them.

[0067] The inferencing phase is accomplished in several passes. In the first pass Arabic and Hebrew letters are assigned the right-to-left direction, while full stops and other alphabetic characters are assigned the left-to-right direction. The next set of passes resolves the directions of digits.

[0068] There are two rules for resolving the direction of Arabic and European numerals. All Arabic numerals are assigned the right-to-left direction. European numerals are assigned the left-to-right direction, unless the European numeral is surrounded by right-to-left characters (Arabic or Hebrew letters), in which case it takes the right-to-left direction. This is accomplished in two passes — a forward pass and a reverse pass.

[0069] The final set of passes resolves the directions of hyphen-minus characters. There are two rules for the resolution of hyphen-minus characters. All hyphen-minus characters become left-to-right, unless the hyphen-minus is surrounded by characters whose direction is right-to-left in which case the hyphen-minus becomes right-to-left. This is the same resolution as digits, but occurs after digit resolution. At this point each character in the domain name has a strong direction.

[0070] According to the preferred embodiment, the resolution of digits and letters makes use of a few simple data structures. A digit accumulator holds a sequence of

European or Arabic numerals. A character stack holds Arabic letters, Hebrew letters, and sequences of digits. A mode variable keeps track of the current direction.

[0071] Further, the preferred embodiment makes use of a few simple operations on these data structures, including a "clear" operation, which outputs each digit from the digit accumulator, then outputs each character from the character stack, and finally outputs the current character. After this operation, the digit accumulator and the character stack are left empty.

[0072] An "empty" operation outputs each character from the character stack, then outputs each digit from the digit accumulator, and finally outputs the current character. After this operation, the digit accumulator and the character stack are empty.

[0073] A "push" operation places the contents of the digit accumulator onto the character stack, and then pushes the current character onto the stack. After this operation the accumulator is empty.

[0074] An "accumulate" operation appends the current character onto the digit accumulator.

[0075] The preferred embodiment of the inventive method for resolving letters and digits is disclosed in pseudo-code in Table 3.

Table 3. Pseudo-code for Resolving Letters and Digits

```
current_character = first character or digit in input_stream
Until end_of_input_stream:
          If (current_character is a left-to-right character) then
                    If (mode = left-to-right) then
                              perform EMPTY
                              else perform CLEAR
                    endif
                    mode = left-to-right
          endIf

          If (current_character is a right-to-left character) AND
                    (current_character is not a digit) then
                    mode = right-to-left
                    perform PUSH
          endIf

          If (current_character is a numeral with right-to-left direction) then
                    mode = right-to-left
                    perform ACCUMULATE
          endIf

          current_character = next character or digit from input stream
```

```
endUntil                /* end of loop to process each char from input stream */

If (mode = left-to-right) then
        perform EMPTY
else
        perform CLEAR
endIf
------------------------------------------------------------------
```

[0076] The bidirectional domain name display method shown in Table 3 converts a string of characters in logical order to a string of the same length in display order. In fact, the method is its own inverse, e.g. $A(A(x))=x$. Hence, this method is a one-to-one function. To further support this rationale, there are several indicators of the behavior of the method which are true of such types of methods. First, it is obvious that the method loses no characters, so the output is a string of the same length as the input string. Second, all left-to-right runs, including full stop and certain hyphen-minus characters, are preserved in exactly their original positions within the resultant string. Third, all right-to-left runs are permuted within their own run. No characters “leak”, “flop” or move to another run and the right-to-left runs are preserved in their same order. Finally, the right-to-left runs are reversed (approximately).

[0077] The nature of reversing right-to-left runs requires further explanation as the numerals (Arabic and European) complicate the matter. Figure 6 shows a sample logical right-to-left string (60), and its corresponding display (61). The output (61) is a string reversal treating digits as units. Hence, this sort of reversal is its own inverse. Therefore, the whole method is its own inverse.

[0078] This method can be used to accommodate two different groups of domain name creators. One group know what they want to register, but are unsure how it will be displayed. On the other hand, there are creators who know what they want to see displayed, but are unsure what logical sequence of characters should be registered. This single universal algorithm addresses both of these situations, which eliminates the need for specialized individual algorithms.

[0079] The disclosed logical methods are preferably realized in software executable by a web server, such as a Java servlet, or by a web client, such as a Java applet. One such embodiment is provided in the Table 4.

[0080] It will be readily apparent to those skilled in the art, though, that alternate embodiments such as coding using alternative software languages and methodologies incorporated into or associated with products such as web browsers, router firmware, etc., may be made without departing from the spirit and scope of the invention. As such, the scope of the invention should be determined by the following claims.

Table 4. Example Embodiment in Java

```
// DomainName.java version 1.0
// Converts domain names in logical and display order.
// Steven Atkin
// 6/15/01

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.LinkedList;
import java.util.Stack;

public class DomainName {

private class AttributedCharacter {
private char character;
private byte direction;
private boolean digit;

public AttributedCharacter (char ch, byte type) {
character = ch;
digit = false;
direction = type;
// set all full stop characters to left
if (type == CS)
direction = L;
else if (type == EN || type == AN)
digit = true;
}
public byte getDir () { return direction; }
public void setDir (byte dir) { direction = dir; }
```

```
31. public boolean isDigit() { return digit; }
32. public char getCharacter() { return character; }
33. }
34.
35. private static final byte L = 0;
36. private static final byte R = 1;
37. private static final byte AL = 2;
38. private static final byte EN = 3;
39. private static final byte ES = 4;
40. private static final byte ET = 5;
41. private static final byte AN = 6;
42. private static final byte CS = 7;
43. private static final byte BN = 8;
44. private static final byte B = 9;
45. private static final byte S = 10;
46. private static final byte WS = 11;
47. private static final byte ON = 12;
48.
49.
:
:
55.
56. // character mappings for 0-127
57. private static final byte[] mixedMap = {
58. BN, BN, BN, BN, BN, BN, BN, BN,
59. BN, S, B, S, WS, B, BN, BN,
60. BN, BN, BN, BN, BN, BN, BN, BN,
61. BN, BN, BN, BN, B, B, B, S,
62. WS, ON, ON, ET, ET, ET, ON, ON,
63. ON, ON, ON, ET, CS, ET, CS, ES,
64. EN, EN, EN, EN, EN, AN, AN, AN,
65. AN, AN, CS, ON, ON, ON, ON, ON,
66. ON, AL, AL, AL, AL, AL, AL, AL,
67. AL, AL, AL, AL, AL, AL, R, R,
68. R, R, R, R, R, R, R, R,
R, R, R, R, R, R, R, S,
ON, L, L, L, L, L, L, L,
L, L, L, L, L, L, L, L,
L, L, L, L, L, L, L, L,
L, L, L, ON, ON, ON, ON, BN
};

private byte[] activeMap = mixedMap;

public DomainName () {
activeMap = mixedMap;
}

// Convert a logical or display domain name
public String convert (String domainName) {
LinkedList attribs = assignAttributes(domainName);

resolveDigits(attribs);
resolveHyphenMinus(attribs);
return reorderStrong(attribs);
}

// Use the character map to get the character attrributes
private LinkedList assignAttributes (String label) {
LinkedList list = new LinkedList();

for (int i = 0; i < label.length(); ++i) {
final char character = label.charAt(i);
final byte type = activeMap[character];
list.add(new AttributedCharacter(character, type));
}
return list;
}


```

```
:
106.
107.
108.
:
112.
113. private String emptyStack(Stack stack) {
114. StringBuffer result = new StringBuffer();
115. while(!stack.empty())
116. result.append(stack.pop());
117. return result.toString();
118. }
119.
120.
121. // Resolve numerals
122. private void resolveDigits (LinkedList label) {
123. byte lastStrong = L;
124. boolean remaining = false;
125. int len = label.size();
126.
127. for (int i = 0; i < len; ++i ) {
128. final byte type = ((AttributedCharacter) label.get(i)).getDir();
129. if (type == L || type == AL || type == R)
130. lastStrong = type;
131. else if (type == EN && lastStrong == L)
132. ((AttributedCharacter) label.get(i)).setDir(L);
133. else if (type == EN)
134. remaining = true;
135. else if (type == AN)
136. ((AttributedCharacter) label.get(i)).setDir(AL);
137. }
138. // If there are any unresolved European numerals, make the second pass.
139. if (remaining) {
140. lastStrong = L;
141. for (int i = len-1; i >= 0; --i) {
```

```
final byte type = ((AttributedCharacter) label.get(i)).getDir();
final boolean isdigit = ((AttributedCharacter) label.get(i)).isDigit();
if ((type == L || type == AL || type == R) && !isdigit)
lastStrong = type;
else if (type == EN && (lastStrong == R || lastStrong == AL))
((AttributedCharacter) label.get(i)).setDir(R);
else if (type == EN)
((AttributedCharacter) label.get(i)).setDir(L);
}
}
}


// Resolve hyphen-minus characters
private void resolveHyphenMinus (LinkedList label) {
byte lastStrong = L;
boolean remaining = false;
int len = label.size();

for (int i = 0; i < len; ++i ) {
final byte type = ((AttributedCharacter) label.get(i)).getDir();
if (type == L || type == AL || type == R)
lastStrong = type;
else if (type == ET && lastStrong == L)
((AttributedCharacter) label.get(i)).setDir(L);
else if (type == ET)
remaining = true;
}
// If there are any hyphen-minus characters left, make the second pass.
if (remaining) {
lastStrong = L;
for (int i = len-1; i >= 0; --i) {
final byte type = ((AttributedCharacter) label.get(i)).getDir();
if (type == L || type == AL || type == R)
lastStrong = type;
```

```
else if (type == ET && (lastStrong == R || lastStrong == AL))
((AttributedCharacter) label.get(i)).setDir(R);
else if (type == ET)
((AttributedCharacter) label.get(i)).setDir(L);
}
}
}

// Reorder the characters once their directions have been resolved
private String reorderStrong (LinkedList attribs) {
byte mode = L;
StringBuffer result = new StringBuffer(attribs.size());
StringBuffer digits = new StringBuffer();
Stack rightStack = new Stack();

for (int i = 0; i < attribs.size(); ++i) {
final char character = ((AttributedCharacter) attribs.get(i)).getCharacter();
final byte dir = ((AttributedCharacter) attribs.get(i)).getDir();
final boolean isdigit = ((AttributedCharacter) attribs.get(i)).isDigit();

// left-to-right characters
if (dir == L) {
if (mode == AL || mode == R) {
result.append(digits);
result.append(emptyStack(rightStack));
}
else {
result.append(emptyStack(rightStack));
result.append(digits);
}
result.append(character);
mode = L;
digits = new StringBuffer();
} // end if left

```

```
// right-to-left characters
else if ((dir == AL || dir == R) && !isdigit) {
rightStack.push(digits);
rightStack.push(new StringBuffer().append(character));
mode = AL;
digits = new StringBuffer();
} // end if Arabic or Hebrew

// Numerals
else if (isdigit && (dir == AL || dir == R)) {
digits.append(character);
mode = dir;
} // end if Arabic or European numeral
} // end for loop

// cleanup
if (mode == R || mode == AL) {
result.append(digits);
result.append(emptyStack(rightStack));
}
else {
result.append(emptyStack(rightStack));
result.append(digits);
}
return result.toString();
}

public static void main (String args[]) {
DomainName domain = new DomainName();
String line = new String();
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

do {
try {
line = in.readLine();
}
catch (IOException e) {
System.out.println("Error on input line");
}
if (line != null && !line.equals(""))
System.out.println(domain.convert(line));
}while (line != null && !line.equals(""));
}
}
```
